John von Neumann Institute for Computing

Towards Improving Skeletons in Eden

M. Hidalgo-Herrero, Y. Ortega-Mallén, F. Rubio

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata
(Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 843-850, 2006.

http://www.fz-juelich.de/nic-series/volume33

Towards Improving Skeletons in Eden*

Mercedes Hidalgo-Herrero[a], Yolanda Ortega-Mallén[b], Fernando Rubio[b]

[a]Dept. Didáctica de las Matemáticas, Universidad Complutense de Madrid, Spain

[b]Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, Spain

The functional language Eden facilitates a skeleton-based methodology for parallel programming. A key point in the efficiency of parallel programs is the distribution of computation among processes. In Eden, this is closely related to its order of evaluation. We describe here an ongoing project whose purpose is to use a prototype implementation of Eden's operational semantics to investigate how alternative evaluation models may either improve or make worse the behavior of the skeletons implemented in Eden.

1. Introduction

The functional parallel language Eden has proven to be highly suitable for a programming methodology based on *algorithmic skeletons*, with the double advantage that skeletons can be implemented and used within the same language. Eden's library provides a rich set of skeletons covering many common parallel patterns such as *parallel map*, *parallel divide-and-conquer*, *parallel search*, and others, as well as typical process topologies like pipelines, grids, rings, and so on [6,14,7,8]. The programmer can either directly use these, or modify them before its use in order to fit better his needs; or even create new skeletons, thus extending the collection.

It is also clear that Eden does not compete for optimal speedups. On the contrary, Eden's strength lies in its higher programming productivity, being its motto: *acceptable speedups at low effort*.

The effectiveness of the use of skeletons depends heavily on the actual implementation of these. Therefore, the majority of skeleton-oriented approaches use low-level languages for the implementation of their skeletons; this should produce accurate and highly efficient implementations, but reduces the flexibility and versatility of the approach, as the set of skeletons usually is fixed. By contrast, Eden offers the possibility of implementing and using skeletons for parallel programming, by considering them as polymorphic higher-order functions. The Eden programmer can choose the process topology and the task granularity, but cannot decide on matters like the placement of processes in processors, or the load balancing strategy. Thus, the efficiency of Eden's skeletons depends on the actual implementation of Eden, that is conditioned by the semantics of the language.

In the present project we desire to investigate alternative semantics for Eden in order to analyze the consequences of some of the decisions adopted during the language design, and in particular how they affect the implementation of skeletons in Eden. For this purpose, it is extremely useful to have a framework where Eden's operational semantics can be easily programmed and that provides mechanisms to reflect changes in the semantics with small effort.

Eden extends the functional language Haskell [13] with coordination features for creating processes with stream-based communication. As a lazy language, Haskell adopts a normal orden of evaluation, avoiding repeated computations by sharing reductions. This lazy approach restricts the exploitation of parallelism because expressions are evaluated only under demand. Therefore, Eden

*Research partially supported by MCyT Spanish project MIDAS: *Metalenguajes para el diseño y análisis integrado de sistemas móviles y distribuidos* (TIC200301000).

overrides the pure lazy approach, combining a non-strict functional application with eager process creation and eager communication. This may produce *speculative* computation, i.e. the calculation of results that may never be used. The amount of speculative computation produced during the evaluation of an Eden program is variable, depending on the number of processors, the speed of basic operations, etc.

The interplay between laziness and eagerness in Eden is precisely established by its operational semantics [3,8]. We are presently developing an interpreter of Eden's operational semantics [4] with the following two main characteristics: (1) different evaluation models for the semantics can be reflected in the implementation with small modifications, and (2) several measures (parallelism, speculative computation, communications) can be taken by modifying some parameters of the semantics.

This interpreter is being implemented in Maude [10,2], a specification language where semantic rules can be represented as rewriting rules; and a *strategy* language [9] is used for controlling the application of the rules.

The rest of the paper is organized as follows. Section 2 gives a brief overview of Eden and its semantics. Section 3 explains the parallel Divide-and-Conquer skeleton and its implementation in Eden. This skeleton is quite simple, but sufficient for showing the kind of analysis that we want to do in the future with more sophisticated skeletons that involve Eden's constructs (streams, dynamic channels, merge process, etc.) that are still to be included in our interpreter. The last section discusses future work.

2. A quick excursion to Eden

We have already mentioned that Eden [1,8] extends the non-strict functional language Haskell with a set of *coordination* features to control parallel evaluation of processes. Coordination in Eden is based on two principal concepts: *explicit definition of processes* and *implicit stream-based communication* [5]. In the same way as there is a distinction between function definition and function application, Eden includes *process abstractions*, i.e. abstract schemes for process behavior, and *process instantiations* for the actual creation of processes.

Moreover, *nondeterminism* is introduced explicitly in Eden by means of a predefined process abstraction which is used to instantiate nondeterministic processes that fairly merge several input streams into a single output stream.

For the purpose of this paper we just concentrate on Eden's essentials, which are captured by the untyped λ-calculus whose abstract syntax is given in Figure 1, where $x \in Var$ represents identifiers and $E \in Exp$ represents expressions.

$E ::= x$		identifier
$\mid$	$\lambda x.E$	λ-abstraction
$\mid$	$E_1 E_2$	application
$\mid$	$E_1 \# E_2$	process instantiation
$\mid$	$\texttt{let}\ \{x_i = E_i\}_{i=1}^{n}\ \texttt{in}\ E$	local declaration

Figure 1. Eden core syntax

When evaluating the expression $E_1 \# E_2$ inside a process p, a new child process q is created which is feeded by its parent process, p, with the value of E_2 via an input channel. Process q evaluates $E_1\ E_2$ and returns the result (to its parent) via an output channel. The diagram in Figure 2 illustrates this behavior.

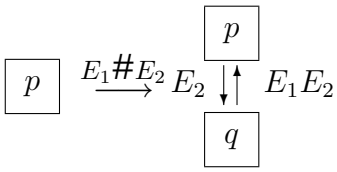

Figure 2. Process creation in Eden

When designing Eden there was great discussion about how to distribute computation between a parent process and its children. In the one extreme the parent would advance as much work as possible, so that every dependent variable of the child process body (or abstraction) should be bound to a *weak head normal form* (whnf) before creating the child process. But this may lead to a poor parallelization, where a process has to do too much computation before delegating work to a subordinate process. In the other extreme —we could say the "laziest"— the parent would pass on all the work to its offspring, so that for an expression $E_1 \# E_2$, the argument E_2 would be evaluated by the parent, while the process abstraction E_1 as well as the application, $E_1\ E_2$, would be evaluated by the new-born child. This may lead to repeated calculations, because certain subexpressions may get evaluated independently by several children of the same parent (as it will illustrated by the example in Section 3). But this can be easily avoided by the programmer, by forcing the evaluation in the parent of these common subexpressions. Therefore, the latter option has been adopted for Eden and its actual implementation, and has been reflected in the operational semantics presented in [8]. Nevertheless, we are interested in analyzing other options; specifically those combinations that are gathered in Figure 3, where EC (*evaluation before copy*) stands for the possibility of evaluating every needed binding before being copied to the initial heap of a newly created process (or the receiver process in the case of a communication); IC (*instantiation copy*) represents the copy of bindings from one process to another corresponding to pending process instantiations; and PAE (*process abstraction evaluation*) indicates the alternatives for the evaluation of a process abstraction in the case of an instantiation: either by the parent process, or by the child.

In the next section we discuss how these combinations affect the behavior of skeletons; in particular, the parallel Divide-and-Conquer scheme.

3. A first case study: Parallel Divide-and-Conquer

As a first case of our analysis we have chosen the parallel Divide-and-Conquer skeleton, a parallelization of the well-known sequential programming scheme. Apart from its simplicity, the main reason for choosing this one is its *task parallel* nature; i.e. it is based on the decomposition of a task into several subtasks to be done in parallel, in contrast to *data parallel* skeletons, where the same operation is applied in parallel to portions of data distributed between processors. The evaluation alternatives for Eden that we have mentioned in the previous section (and that are summarized in

	EC	IC	PAE
1	yes	yes	parent
2	yes	no	parent
3	yes	yes	child
4	yes	no	child
5	no	yes	child
6	no	no	child

Figure 3. Evaluation alternatives

Figure 3) should have a bigger impact on task parallel skeletons, because they affect the work to be done by each process.

Another reason for starting with this skeleton is that it can be easily implemented by creating a dynamic tree of processes where each process is connected to its parent. This hierarchical topology matches perfectly with the process creation and communication mechanisms in Eden, while other skeletons involve special topologies (like pipelines, rings or grids) that are far better implemented in Eden by using dynamic channels[2].

Next we express the Divide-and-Conquer scheme in the restricted syntax given in Figure 1. First we give the sequential version for a split into two subproblems; afterwards we show a straightforward parallel version where every subproblem causes the creation of a process to resolve it.

```
dc = (\trivial.(\solve.(\split.(\combine.(\x.
 (let
   subpr = (split x)
   sol1 = (((((dc trivial) solve) split) combine) (fst subpr))
   sol2 = (((((dc trivial) solve) split) combine) (snd subpr))
  in  (cond (iszero (trivial x))
        (((combine x) sol1) sol2)
        (solve x))
 ))))))


dc_par = (\trivial.(\solve.(\split.(\combine.(\x.
 (cond (iszero (trivial x))
   (let
     subpr = (split x)
     sol1 = (((((dc_par trivial) solve) split) combine) # (fst subpr))
     sol2 = (((((dc_par trivial) solve) split) combine) # (snd subpr))
    in (((combine x) sol1) sol2))
   (solve x))
 )))))
```

In practice, it is more efficient to stop the parallel unfolding before trivial cases are reached; therefore we also present a version where a parameter `depth` determines the maximum level allowed for creating children.

[2]Even though it is possible to create the same topologies without using dynamic channels [12], it is much more complicated and inefficient.

```
dc_par_lim = (\depth.(\trivial.(\solve.(\split.(\combine.(\x.
 (cond (iszero depth)
   (((((dc2 trivial) solve) split) combine) x)
   (cond (iszero (trivial x))
     (let
       subpr = (split x)
       d1 = ((sub depth) one)
       sol1 = (((((dc_par_lim d1 trivial) solve) split) combine) # (fst subpr))
       sol2 = (((((dc_par_lim d1 trivial) solve) split) combine) # (snd subpr))
      in (((combine x) sol1) sol2))
     (solve x)))
 ))))))
```

In the coding we have used functions like `cond` (conditional), `fst` (first) and `snd` (second) for extracting tuple components, `one`, `zero`, `iszero` and `sub` (subtract) that can be easily defined in a λ-calculus (see for instance [11]).

3.1. Discussion on evaluation alternatives

One key design decision for Eden was how to deal with free variables occurring in expressions that should be exported from one process to another —for instance, when creating a new process, or when communicating abstractions—. The alternatives (column EC in Figure 3) are either to evaluate these variables before being copied, or to copy the corresponding evaluation subgraph in the "receiver". The latter may lead to repeat some computations, as it is illustrated with the help of the Divide-and-Conquer skeletons given above. Let us concentrate on the subexpressions

```
(((((dc_par trivial) solve) split) combine) # (fst subpr))
(((((dc_par trivial) solve) split) combine) # (snd subpr))
```

and

```
((((((dc_par_lim depth_1) trivial) solve) split) combine) # (fst subpr))
((((((dc_par_lim depth_1) trivial) solve) split) combine) # (snd subpr))
```

contained in `dc_par` and `dc_par_lim` respectively. The variables `depth_1`, `trivial`, `solve`, `split`, and `combine` are free in each subprocess abstraction. If they are required to be evaluated before the creation of a subprocess, then the evaluation of all these variables is carried out by the parent process and it is done only once. However, if the evaluation is left to the children, the computation is done twice.

On the other hand, the process abstractions

```
((((dc_par trivial) solve) split) combine)
```

and

```
(((((dc_par_lim depth_1) trivial) solve) split) combine)
```

might be bound to a variable in the corresponding let-expression. The relevance of this modification depends on the semantics of the language. The options are gathered in Figure 4; let us analyze them:

1. In this first case the amount of repeated computation is minimized because the abstraction is evaluated once.

2. The evaluation of the abstraction is repeated by each child. In this case it does not matter whether the abstraction is bound to a variable or not.

3. Since the abstraction is not bound to a variable it will be evaluated twice by the parent process.

	Bound to variable	Process abstraction evaluation
1	Yes	Parent
2	Yes	Child
3	No	Parent
4	No	Child

Figure 4. Process abstraction binding and evaluation

4. The evaluation in this last case coincides with that of the second case.

The evaluation of a process abstraction either by the parent or by the children may influence in the speed of evaluation. If the process is created without waiting for the evaluation of the abstraction, then this can be evaluated simultaneously with the expression corresponding to the child input channel, i.e. `(fst subpr)` in our example.

We finish our discussion by remarking that in this example it is irrelevant whether instantiations are copied from one process to another or not (column IC in Figure 3) because there is no interdependence between the abstractions `sol1` and `sol2`.

4. Future work

We have given a first step in our project to analyze the way skeletons are affected by different semantics choices for Eden. However, this analysis has only been made from a theoretical point of view because the actual implementation of Eden's semantics in Maude presents some problems of efficiency that impede us, at the moment, to extract useful measures for our examples. Therefore, our work is still in a preliminary state.

We intend to make a thorough analysis of the different semantic approaches by executing programs based on the skeletons presented in [7]. In order to develop this task, the kernel of Eden presented in this paper needs to be extended. First of all, we have to include streams to represent unbounded communication channels; afterwards, we shall consider dynamic channels, in order to specify more directly non-hierarchical topologies. Finally, we will introduce non-determinism for expressing many-to-one communication, that is essential in many parallel applications, like for instance, in clients-server models.

The analysis will be based on measures of computation such as the number of computation steps, the amount of communication carried out, the number of processes in the final system or the (maximal) amount of thread parallelism that are already included in our Eden interpreter implemented in Maude. In the Divide-and-Conquer skeleton it is easy to work out the number of processes that are created. However, some other measures cannot be obtained unless the computation is effectively made. These measures will help us to analyze the advantages and drawbacks of each semantical option and elucidate whether there is an optimal approach.

References

[1] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. Eden: Language definition and operational semantics. Technical Report 96/10, Reihe Informatik, FB Mathematik, Philipps-Universität Marburg, Germany, URL `http://www.mathematik.uni-marburg.de/~eden/`, 1996.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude Manual (Version 2.1)*, March 2004. URL `http://maude.cs.uiuc.edu/manual`.

[3] M. Hidalgo-Herrero and Y. Ortega-Mallén. An operational semantics for the parallel language Eden. *Parallel Processing Letters (World Scientific Publishing Company)*, 12(2):211–228, 2002.

[4] M. Hidalgo-Herrero, Alberto Verdejo, and Y. Ortega-Mallén. Looking for eden through maude and its strategies. In *Proc. of the 5th Jornadas sobre Programacin y Lenguajes*, 2005. To appear.

[5] G. Kahn and D. MacQueen. Coroutines and networks of parallel processes. In *IFIP'77*, pages 993–998. Eds. B. Gilchrist. North-Holland, 1977.

[6] Ulrike Klusik, Rita Loogen, Steffen Priebe, and Fernando Rubio. Implementation skeletons in eden: Low-effort parallel programming. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages, (IFL'00 selected papers)*, pages 71–88. LNCS 2011, Springer, 2001.

[7] R. Loogen, Y. Ortega-Mallén, R. Pea, S. Priebe, and F. Rubio. *Patterns and Skeletons for Parallel and Distributed Computing*, chapter 4: Parallelism Abstractions in Eden, pages 95–128. Eds. F. A. Rabhi and S. Gorlatch. Springer, 2002.

[8] R. Loogen, Y. Ortega-Mallén, and R. Peña. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[9] N. Martí-Oliet, J. Meseguer, and A. Verdejo. Towards a strategy language for Maude. In N. Martí-Oliet, editor, *Proceedings Fifth International Workshop on Rewriting Logic and its Applications, WRLA 2004, Barcelona, Spain, March 27 – April 4, 2004*, volume 117 of *Electronic Notes in Theoretical Computer Science*, pages 417–441. Elsevier, 2004.

[10] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[11] G. Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Addison-Wesley, 89.

[12] R. Peña, F. Rubio, and C. Segura. Deriving non-hierarchical process topologies. In *Trends in Functional Programming (Selected papers of the 3rd Scottish Functional Programming Workshop)*, volume 3, pages 51–62. Intellect, 2002.

[13] S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[14] F. Rubio. *Programación funcional paralela eficiente en Eden*. PhD thesis, Dept. Sistemas Informáticos y Programación, Universidad Complutense de Madrid, 2001.

850